Hangman

Genric_vector code was provided by Umass lowell. All other files where constructed by scratch for the evil hangman game and to practice C concepts.

MakeFile

```
CC = gcc

CFLAGS = -g -Wall --std=c99 -Wfatal-errors -Wno-unused-variable

OBJECTS = main.o my_string.o generic_vector.o hangman.o avl_tree.o

OBJECT2 = my_string.o unit_test.o test_def.o generic_vector.o

ALL = main.o my_string.o generic_vector.o unit_test.o test_def.o hangman.o avl_tree.o

string_driver: $(OBJECTS)

	$(CC) $(CFLAGS) -o string_driver $(OBJECTS)

main.o: main.c

	$(CC) $(CFLAGS) -c main.c -o main.o

hangman.o: hangman.c hangman.h

	$(CC) $(CFLAGS) -c hangman.c -o hangman.o

generic_vector.o: generic_vector.c generic.h generic_vector.h

	$(CC) $(CFLAGS) -c generic_vector.c -o generic_vector.o

unit_test.o: unit_test.c unit_test.h

	$(CC) $(CFLAGS) -c unit_test.c -o unit_test.o

test_def.o: test_def.c

	$(CC) $(CFLAGS) -c test_def.c -o test_def.o

my_string.o: my_string.c my_string.h status.h

	$(CC) $(CFLAGS) -c my_string.c  -o my_string.o

avl_tree.o: avl_tree.c avl_tree.h

	$(CC) $(CFLAGS) -c avl_tree.c -o avl_tree.o

unit_test: $(OBJECT2)

	$(CC) $(CFLAGS) -o unit_test $(OBJECT2)

clean:  rm -f $(ALL)
```

Hangman.h

```
#ifndef HANGMAN_H

#define HANGMAN_H


#include "my_string.h"

#include "generic_vector.h"

#include "avl_tree.h"

#include "generic.h"

#include "status.h"


//returns an array

GENERIC_VECTOR* hangman_set_up(GENERIC_VECTOR* hVector, int size);


//pre: takes a valid item handle, a valid vector, and the valid size

//Status fill_vector_to_size(GENERIC_VECTOR hVector, IT);


void display_key(MY_STRING current_word_family);


//runs one play through of the game and returns a continuation value

int hangman_game_loop(GENERIC_VECTOR hVector[]);


#endif
```

Hangman.c

```
#include "hangman.h"

#include "ctype.h"

void clear_keyboard_buffer(void);

void check_word_family(MY_STRING current_word_family);


GENERIC_VECTOR *hangman_set_up(GENERIC_VECTOR *array_hVector, int size)

{

	printf("Loading...\n");
	FILE *fp;
	int i;

	AVL_TREE tree;
	fp = fopen("dictionary.txt", "r");
	ITEM hItem = my_string_init_default();

	for (i = 0; i < size; i++)
	{
		array_hVector[i] = generic_vector_init_default(my_string_assignment, my_string_destroy);
	}

	while (my_string_extraction(hItem, fp))
	{
		generic_vector_push_back(array_hVector[my_string_get_size(hItem)], hItem);
	}

	my_string_destroy(&hItem);
	fclose(fp);

	return *array_hVector;
}

int hangman_game_loop(GENERIC_VECTOR words_by_length[])
{
	int noc;
	int word_length;
	int guesses_remaining;
	int guess_count = 0;
	int vecSize = 0;
	int i;
	char c;
	AVL_TREE largest_node;
	GENERIC_VECTOR current_words = NULL ;
	MY_STRING current_word_family = NULL;
	MY_STRING new_key = NULL;
	Status status;
	char guessed_chars[90];



	while (1)
	{
		printf("Please pick a valid word length to play with: ");
		noc = scanf("%d", &word_length);
		clear_keyboard_buffer();

		if (noc == 1 && word_length > 0 && word_length < 30 &&
generic_vector_get_size(words_by_length[word_length])) {
			break;
		}
	}

	current_words = generic_vector_make_clone(words_by_length[word_length]);
	if(current_words == NULL)
	{
		return -3;
	}
	current_word_family = my_string_init_default();
	if (current_word_family == NULL) {
		return -4;
	}
	for (i = 0; i < word_length; i++)
	{
		if (my_string_push_back(current_word_family, '-') == FAILURE) {
			return -7;
```

```
                }
        }

        //guess loop
        while(1)
        {
                printf("How many guesses would you like: ");
                noc = scanf("%d", &guesses_remaining);
                clear_keyboard_buffer();
         if(noc == 1 && guesses_remaining > 0)
                {
                        break;
                }
        }

         int hints;
    while(1)
        {
        char response;
         printf("Do you want hints y/n: ");
         noc = scanf(" %c", &response);
         response = tolower(response);
                if(noc == 1 && (response == 'y' || response == 'n'))
                {
                        switch(response){
                                case 'y' :
                                  hints = 1;
                                  break;
                                default:
                                  hints = 0;
                                  break;
                        }
```

```
                    break;
            }
    }



for(i = 0; i < 90; i++)
{
            guessed_chars[i] = '0';
}

//main game loop
while (guesses_remaining != 0)
{

            printf("You have %d guesses left!\n", guesses_remaining);
            printf("Used letters: %.*s\n", guess_count, guessed_chars);

            //display chosen word
            display_key(current_word_family);

            //validation loop
            while(1)
            {
                    printf("Please enter your guess: ");
                    noc = scanf(" %c", &c);
                    clear_keyboard_buffer();
                    c = tolower(c);
                    i = 0;
                    while(guessed_chars[i] != '\0')
                    {
                            if(guessed_chars[i] == c)
                            {
```

```
                noc = -1;
            }
            i++;
        }
        if(noc == 1 && isalpha(c))
        {
            break;
        }
}

AVL_TREE key_closet = avl_tree_init_default();
for (i = 0; i < generic_vector_get_size(current_words); i++)
{
        new_key = my_string_init_default();
        MY_STRING word = *generic_vector_at(current_words, i);

        //gets new value
        if (get_word_key_value(current_word_family, new_key, word, c) == FAILURE)
        {
            return -5;
        }

        //fill key closet
        if (avl_tree_insert(&key_closet, new_key, word, hints) == FAILURE)
        {
            printf("avl_tree_insert failed\n");
            return -6;
        }
}

//finds largest vector in tree using key values then destroys tree once vec is found
largest_node = avl_get_largest_node(key_closet);
```

```
generic_vector_destroy(&current_words);
avl_get_key_take_value(largest_node, &current_word_family, &current_words);
avl_tree_destroy(&key_closet);

//handles guess count
for (i = 0; i < my_string_get_size(current_word_family); i++)
{
        if (*my_string_at(current_word_family, i) == c)
        {
                guesses_remaining++;
                break;
        }
}
guessed_chars[guess_count++] = c;
guesses_remaining--;

int flag = 0;
for(i = 0; i < my_string_get_size(current_word_family); i++)
{
        if(*(my_string_at(current_word_family, i))== '-')
        {
                flag = 1;
            break;
        }
}
if(flag == 0)
{
        printf("YOU WON!\n");
        printf("The word was ");
        display_key(*generic_vector_at(current_words,0));
        break;
}
```

```
                        if(guesses_remaining == 0)

                        {

                                    printf("You are out of choices!\n");

                                    printf("The word was ");

                    display_key(*generic_vector_at(current_words,rand() %
generic_vector_get_size(current_words)));

                                    printf("GAME OVER!\n");

                                    break;

                        }

                        if(hints)

                        {

                                    printf("choices left: %d\n", generic_vector_get_size(current_words));

                        }

            }



            my_string_destroy(&current_word_family);

            generic_vector_destroy(&current_words);

            return 1;

}


//will display the

void display_key(MY_STRING current_word_family)

{

            //int i = my_string_get_size(current_word_family);


            printf("%s\n", my_string_c_str(current_word_family));

}



void clear_keyboard_buffer(void)

{
            char c;

            scanf("%c", &c);

            while (c != '\n')

            {

                        scanf("%c", &c);

            }

}
```

AVL_TREE.H

```
#ifndef AVL_TREE_H

#define AVL_TREE_H

#include "generic_vector.h"

#include "my_string.h"


typedef void* AVL_TREE;




//pre:is given a valid generic vector

//post: returns a valid avl_tree handle, returns Null if failure

AVL_TREE avl_tree_init_default();


//pre: is past a valid handle to an avl tree, and is passed a vaild handle to

// to a genric vector

//post: returns success if node is added to proper part of hTree, and

//hdata is succesfuly stored in node in proper location

//returns failure if there is memory issues

Status avl_tree_insert(AVL_TREE *phTree, MY_STRING new_key, MY_STRING word, int output);


//pre past a vailid tree MY_string key:

//post: returns address of largest node

AVL_TREE avl_get_largest_node(AVL_TREE hTree);


void avl_get_key_take_value(AVL_TREE hTree, MY_STRING *phKey, GENERIC_VECTOR *phVec);

//will destroy tree
```

```
void avl_tree_destroy(AVL_TREE *phTree);


#endif
```

AVL_TREE.C

```
#include "avl_tree.h"

#include "generic_vector.h"

#include "stdlib.h"

#include "my_string.h"


typedef struct node Node;


struct node {

        GENERIC_VECTOR data;

        MY_STRING key;

        Node *left;

        Node *right;

        int BF;

};


int avl_tree_balance(Node *pNode);

Status left_rotation(Node *pNode);

Status right_rotation(Node *pNode);


Node *make_node(MY_STRING new_key, MY_STRING word)

{

        Node *pNode = (Node *)malloc(sizeof(Node));

        if (pNode == NULL)

        {

                printf("ERROR: Failed to allocate space\n");

                return NULL;

        }

        pNode->data = generic_vector_init_default(my_string_assignment, my_string_destroy);

        if (pNode->data == NULL)
```

```
	{
		free(pNode);
		return NULL;
	}

	if (generic_vector_push_back(pNode->data, word) == FAILURE)
	{
		free(pNode);
		return NULL;
	}

	// if you want to be awful: pNode->left = pNode->right = NULL;
	pNode->left = NULL;
	pNode->right = NULL;

	pNode->key = new_key;
	pNode->BF = 0;
	return pNode;
}

AVL_TREE avl_tree_init_default()
{
	return NULL;
}

Status avl_tree_insert(AVL_TREE *phTree, MY_STRING new_key, MY_STRING word, int output)
{
	Node **ppNode = (Node **)phTree;
	Node *pNode = *ppNode;

	if (pNode == NULL)
	{
		*ppNode = make_node(new_key, word);
		if(output)
		{
			printf("%s\n", my_string_c_str(new_key));
		}
		return *ppNode == NULL ? FAILURE : SUCCESS;
	}

	int const compare = my_string_compare(new_key, pNode->key);
	if (compare < 0)
	{
		return avl_tree_insert((AVL_TREE *)(&pNode->left), new_key, word, output);
		//avl_tree_balance(pNode);
	}
	else if (compare > 0)
	{
		return avl_tree_insert((AVL_TREE *)(&pNode->right), new_key, word, output);
		//avl_tree_balance(pNode);
	}
	else
	{
		//node already exists, add word to the node's values
		if (pNode->data == NULL)
		{
			pNode->data = generic_vector_init_default(my_string_assignment,
my_string_destroy);
			if (pNode->data == NULL)
			{
				return FAILURE;
			}
		}
		generic_vector_push_back(pNode->data, word);
		//no new node was created, so destroy the key
```

```
                my_string_destroy(&new_key);
                return SUCCESS;
        }
}

static void get_largest_node(Node *pNode, int *max_size, Node **max_node)
{
        if (pNode == NULL) {
                return;
        }

        int const size = generic_vector_get_size(pNode->data);
        if (*max_node == NULL || *max_size < size)
        {
                *max_size = size;
                *max_node = pNode;
        }

        get_largest_node(pNode->left, max_size, max_node);
        get_largest_node(pNode->right, max_size, max_node);
}

AVL_TREE avl_get_largest_node(AVL_TREE hTree) {
        Node *pNode = (Node *)hTree;
        Node *max_node = NULL;
        int max_size;

        get_largest_node(pNode, &max_size, &max_node);
        return max_node;
}

int my_max(int a, int b)
```

```
{
        if (a > b)
        {
                return a;
        }
        return b;
}

int tree_report_height(Node *root)
{
        if (root == NULL)
        {
                return 0;
        }
        else
        {
                return 1 + my_max(tree_report_height(root->left), tree_report_height(root->right));
        }
}


//needs to find bottom node then travel back up it unsureing everything is balanced
//so count must start at the bottom of recursion
int avl_tree_balance(Node *pRoot)
{

        int heightLeft = tree_report_height(pRoot->left);
        int heightRight = tree_report_height(pRoot->right);
        pRoot->BF = heightRight - heightLeft;

        if (pRoot->BF < -1)
        {
```

```
		right_rotation(pRoot);
	}
	else if (pRoot->BF > 1)
	{
		left_rotation(pRoot);
	}

	return 0;
}

Status left_rotation(Node *pNode)
{

	if (pNode->right != NULL)
	{
		if (pNode->right->BF < 0)
		{
			right_rotation(pNode->right);
		}
	}
	Status status = FAILURE;
	Node *temp1, *temp2, *orphan = NULL;
	temp1 = (Node *)malloc(sizeof(Node));
	if (temp1 != NULL)
	{
		//make a temp
		status = SUCCESS;
		temp1->BF = 0;
		temp1->data = pNode->data;
		temp1->left = pNode->left;
		temp1->right = pNode->right;
		temp1->key = pNode->key;
		//assign orphan if it exsits & cut it from tree
		if (pNode->right != NULL && pNode->right->left != NULL)
		{
			orphan = pNode->right->left;
			pNode->right->left = NULL;
		}

		//set right child as new parent
		temp2 = pNode->right;
		if (temp2 != NULL)
		{
			pNode->data = pNode->right->data;
			pNode->key = pNode->right->key;
			pNode->right = pNode->right->right;

			free(temp2);
		}
		pNode->left = temp1;
		//re-assign orphan to the former parent
		temp1->right = orphan;

		//get new balance factor
		pNode->BF = 0;//fix
	}

	return status;
}

Status right_rotation(Node *pNode)
{
	if (pNode->left != NULL)
```

```
{
        if (pNode->left->BF > 0)
        {
                left_rotation(pNode->left);
        }
}
Status status = FAILURE;
Node *temp1, *temp2, *orphan = NULL;
temp1 = (Node *)malloc(sizeof(Node));
if (temp1 != NULL)
{
        //make a temp
        status = SUCCESS;
        temp1->BF = 0;
        temp1->data = pNode->data;
        temp1->left = pNode->left;
        temp1->right = pNode->right;
        temp1->key = pNode->key;

        //assign orphan if it exsits & cut it from tree
        if (pNode->left != NULL && pNode->left->right != NULL)
        {
                orphan = pNode->left->right;
                pNode->left->right = NULL;
        }

        //set left child as new parent
        temp2 = pNode->left;
        if (temp2 != NULL)
        {
                pNode->data = pNode->left->data;
                pNode->key = pNode->left->key;
```

```
				pNode->left = pNode->left->left;
				pNode->right = temp1;
				free(temp2);
			}

			//re-assign orphan to the former parent
			temp1->left = orphan;

			//get new balance factor
			pNode->BF = 0;//fix
		}

		return status;
}

GENERIC_VECTOR avl_tree_swap_destroy(MY_STRING *current_word_family, AVL_TREE key_closet,
AVL_TREE next_vector)
{
		int i;
		Node *pTree = (Node *)next_vector;
		MY_STRING temp;
		GENERIC_VECTOR newVector = generic_vector_init_default(my_string_assignment,
my_string_destroy);
		MY_STRING new_key = my_string_init_default();

		my_string_assignment(current_word_family, pTree->key);

		for (i = 0; i < generic_vector_get_size(pTree->data); i++)
		{
			generic_vector_push_back(newVector, *(generic_vector_at(pTree->data, i)));
		}
```

```
        avl_tree_destroy(key_closet);

        return newVector;
}

void avl_get_key_take_value(AVL_TREE hTree, MY_STRING *phKey, GENERIC_VECTOR *phVec)
{
        Node *pNode = (Node *)hTree;
        my_string_assignment(phKey, pNode->key);
        *phVec = pNode->data;
        pNode->data = NULL;
}

static void destroy_nodes(Node *pNode) {
        if (pNode == NULL)
        {
                return;
        }

        destroy_nodes(pNode->left);
        destroy_nodes(pNode->right);

        generic_vector_destroy(&(pNode->data));
        my_string_destroy(&(pNode->key));

        free(pNode);
}

void avl_tree_destroy(AVL_TREE *phTree)
{
        Node *pNode = (Node *)*phTree;
        destroy_nodes(pNode);

        *phTree = NULL;

}
```

MY_STRING.H

```
#ifndef MY_STRING_H

#define MY_STRING_H



#include <stdio.h>

#include <stdlib.h>

#include "generic.h"

#include "status.h"



typedef void* MY_STRING;


//Precondition: None

//Postcondition: Allocate space for a string object that represents the empty

// string. The string will have capacity 7 and size 0 by default. A copy of

// the address of the opaque object will be returned on success and NULL on

// failure.

MY_STRING my_string_init_default(void);


//Precondition: pLeft is the address of a MY_STRING handle

// containing a valid MY_STRING object address OR NULL.

// The value of Right must be the handle of a valid MY_STRING object

//Postcondition: On Success pLeft will contain the address of a handle

// to a valid MY_STRING object that is a deep copy of the object indicated

// by Right. If the value of the handle at the address indicated by

// pLeft is originally NULL then the function will attempt to initialize

// a new object that is a deep copy of the object indicated by Right,

// otherwise the object indicated by the handle at the address pLeft will
// attempt to resize to hold the data in Right. On failure pLeft will be
// left as NULL and any memory that may have been used by a potential
// object indicated by pLeft will be returned to the freestore.
Status my_string_assignment(ITEM* phLeft, ITEM hRight);

//Precondtion: c_string is a valid null terminated c-string.
//Postcondition: Allocate space for a string object that represents a string
// with the same value as the given c-string. The capacity of the string
// object will be set to be one greater than is required to hold the string.
// As an example, the string "the" would set capacity at 4 instead of 3. A
// copy of the address of the opaque object will be returned on success and
// NULL on failure.
MY_STRING my_string_init_c_string(const char* c_string);

//Precondition: hMy_string is the handle of a valid My_string object.
//Postcondtion: Returns a copy of the integer value of the object's capacity.
int my_string_get_capacity(MY_STRING hMy_string);

//Precondition: hMy_string is the handle of a valid My_string object.
//Postcondtion: Returns a copy of the integer value of the object's size.
int my_string_get_size(MY_STRING hMy_string);

//Precondition: hLeft_string and hRight_string are valid My_string objects.
//Postcondition: returns an integer less than zero if the string represented
// by hLeft_string is lexicographically smaller than hRight_string. If
// one string is a prefix of the other string then the shorter string is
// considered to be the smaller one. (app is less than apple). Returns
// 0 if the strings are the same and returns a number greater than zero
// if the string represented by hLeft_string is bigger than hRight_string.
int my_string_compare(MY_STRING hLeft_string, MY_STRING hRight_string);

//Precondition: hMy_string is the handle to a valid My_string object.
```

```
//Postcondition: hMy_string will be the handle of a string object that contains
// the next string from the file stream fp according to the following rules.
// 1) Leading whitespace will be ignored.
// 2) All characters (after the first non-whitespace character is obtained
// and included) will be added to the string until a stopping condition
// is met. The capacity of the string will continue to grow as needed
// until all characters are stored.
// 3) A stopping condition is met if we read a whitespace character after
// we have read at least one non-whitespace character or if we reach
// the end of the file.
// Function will return SUCCESS if a non-empty string is read successfully.
// and failure otherwise. Remember that the incoming string may aleady
// contain some data and this function should replace the data but not
// necessarily resize the array unless needed.
Status my_string_extraction(MY_STRING hMy_string, FILE* fp);
//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Writes the characters contained in the string object indicated
// by the handle hMy_string to the file stream fp.
// Function will return SUCCESS if it successfully writes the string and
// FAILURE otherwise.
Status my_string_insertion(MY_STRING hMy_string, FILE* fp);

//Pre-condition: gets a handle to a string and value to add to the string
//Post-condition: item will be added to the string and string->data will be double if
//the data is not within the capacity
Status my_string_push_back(MY_STRING hMy_string, char value);

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: If successful, places the character item at the end of the
// string and returns SUCCESS. If the string does not have enough room and
// cannot resize to accomodate the new character then the operation fails
// and FAILURE is returned. The resize operation will attempt to amortize
// the cost of a resize by making the string capacity somewhat larger than
// it was before (up to 2 times bigger).
Status my_string_push_back(MY_STRING hMy_string, char item);

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Removes the last character of a string in constant time.
// Guaranteed to not cause a resize operation of the internal data. Returns
// SUCCESS on success and FAILURE if the string is empty.
Status my_string_pop_back(MY_STRING hMy_string);

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Returns the address of the character located at the given
// index if the index is in bounds but otherwise returns NULL. This address
// is not usable as a c-string since the data is not NULL terminated and is
// intended to just provide access to the element at that index.
char* my_string_at(MY_STRING hMy_string, int index);

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Returns the address of the first element of the string object
// for use as a c-string. The resulting c-string is guaranteed to be NULL
// terminated and the memory is still maintained by the string object though
// the NULL is not actually counted as part of the string (in size).
char* my_string_c_str(MY_STRING hMy_string);

//precondition: hMystring is a valid object handle
//postcondtion: the memory footprint of hMystring data will be doubled
Status my_string_resize_up(MY_STRING hMy_string);

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Returns an enumerated type with value TRUE if the string
// is empty and FALSE otherwise.
Boolean my_string_is_empty(MY_STRING hMy_string);
```

```
//Precondition: hResult and hAppend are handles to valid My_string objects.
//Postcondition: hResult is the handle of a string that contains the original
// hResult object followed by the hAppend object concatenated together. This
// function should guarantee no change to the hAppend object and return
// SUCCESS if they operation is successful and FAILURE if the hResult object
// is unable to accomodate the characters in the hAppend string perhaps
// because of a failed resize operation. On FAILURE, no change to either
// string should be made.
Status my_string_concat(MY_STRING hResult, MY_STRING hAppend);


//Precondition:current_word_family, new_key and word are all handles to valid
// MY_STRING opaque objects. guess is an alphabetical character that can be either
// upper or lower case.
//Postcondition: Returns SUCCESS after replacing the string in new_key with the key
// value formed by considering the current word family, the word and the guess.
// Returns failure in the case of a resizing problem with the new_key string.
Status get_word_key_value(MY_STRING current_word_family, MY_STRING new_key, MY_STRING word,
char guess);


//Precondition: phMy_string holds the address of a valid handle to a MY_STRING
// object.
//Postcondition: The memory used for the MY_STRING object has be reclaimed by
// the system and the handle referred to by the pointer phMy_string has been
// set to NULL.
void my_string_destroy(ITEM* phMy_string);


#endif
```

My_STRING.C

```
//file contains main.c functions and


#include "my_string.h"
```

```
struct my_string
{
	int size;
	int capacity;
	char *data;
};

typedef struct my_string My_string;


Status my_string_assignment(ITEM *phLeft, ITEM hRight)
{
	int i;
	char *temp;
	My_string *pLeft = (My_string *)*phLeft;
	My_string *pRight = (My_string *)hRight;

	if (pLeft == NULL)
	{
		//we need to create the left my_string

		pLeft = (My_string *)malloc(sizeof(My_string));
		if (pLeft == NULL)
		{
			printf("Failed to create object.\n");
			return FAILURE;
		}
		*phLeft = pLeft;

		pLeft->size = pRight->size;
		pLeft->capacity = pRight->capacity;
		pLeft->data = (char *)malloc(sizeof(char) * pRight->capacity);
		if (pLeft->data == NULL) {


			free(pLeft);
			return FAILURE;


		}
		for (i = 0; i < pLeft->size; i++)
		{
			pLeft->data[i] = pRight->data[i];
		}
		//printf("%s\n", pLeft->data);//delete
	}
	else if (pRight->size >= pLeft->capacity)
	{
		//we need to resize the left my_string


		pLeft->size = pRight->size;
		pLeft->capacity = pRight->capacity;
		temp = (char *)malloc(sizeof(char) * pRight->capacity);
		if (temp == NULL)
		{
			return FAILURE;
		}
		for (i = 0; i < pRight->size; i++)
		{
			temp[i] = pRight->data[i];
		}
		free(pLeft->data);
		pLeft->data = temp;
	}
	else
```

```
		{
			//just copy the data and members


			pLeft->size = pRight->size;


			for (i = 0; i < pRight->size; i++)
			{
				pLeft->data[i] = pRight->data[i];
			}
		}


		return SUCCESS;
}


MY_STRING my_string_init_default(void)
{
		//allocates space for the structure if it fails null returned exit code 1
		My_string *pstring = (My_string *)malloc(sizeof(My_string));
		if (pstring != NULL)
		{
			//assigns default behavior of My_string
			pstring->size = 0;
			pstring->capacity = 7;
			pstring->data = (char *)malloc(sizeof(char) * pstring->capacity);//allocates space for char
string and stores the address in data pointer
			if (pstring->data == NULL)
			{
				//if allocation failed then the programs frees up allocated space and returns null
				free(pstring);
				return NULL;


			}
		}
```

```
	return (MY_STRING)pstring;


}


void my_string_destroy(ITEM *phMy_string)

{

	//function is past a void* holding the address of the structure, then its casted to know type so
adress can be acessesd

	My_string *pString = (My_string *)*phMy_string;

	if (pString == NULL) {

		return;

	}


	//gets rid of dynamicaly allocated arrray for data

	free(pString->data);

	free(pString);

	*phMy_string = NULL;


	//printf("Process complete\n");//confirmation statement




}


//Precondtion: c_string is a valid null terminated c-string.

//Postcondition: Allocate space for a string object that represents a string

// with the same value as the given c-string. The capacity of the string

// object will be set to be one greater than is required to hold the string.

// As an example, the string "the" would set capacity at 4 instead of 3. A

// copy of the address of the opaque object will be returned on success and

// NULL on failure.

MY_STRING my_string_init_c_string(const char *c_string)

{

	int i = 0;
	//allocates space for object
	My_string *p_string = (My_string *)malloc(sizeof(My_string));
	if (p_string != NULL)
	{
		while (c_string[i] != '\0')
		{
			i++;
		}
		//size starts as given
		p_string->size = i;
		p_string->capacity = i + 1;//capacity is one larger than size
		p_string->data = (char *)malloc(sizeof(char) * p_string->capacity);//mallocs a dynamic array
for the string
		if (p_string->data == NULL)//safty net
		{
			free(p_string->data);
			return NULL;
		}
		//copy loop
		for (i = 0; i < p_string->size; i++)
		{
			(*p_string).data[i] = c_string[i];
		}



	}
	//printf("copy complete\n");//confirmation
	return p_string;


}


//Precondition: hMy_string is the handle of a valid My_string object.
//Postcondtion: Returns a copy of the integer value of the object's capacity.
```

```
int my_string_get_capacity(MY_STRING hMy_string)

{

        My_string *p_string = (My_string *)hMy_string;//cast to the known type

        //return an integer value

        return p_string->capacity;


}


//Precondition: hMy_string is the handle of a valid My_string object.

//Postcondtion: Returns a copy of the integer value of the object's size.

int my_string_get_size(MY_STRING hMy_string)

{

        My_string *p_string = (My_string *)hMy_string;//cast to the known type

        //return an integer value

        return p_string->size;

}



//Precondition: hLeft_string and hRight_string are valid My_string objects.

//Postcondition: returns an integer less than zero if the string represented

// by hLeft_string is lexicographically smaller than hRight_string. If

// one string is a prefix of the other string then the shorter string is

// considered to be the smaller one. (app is less than apple). Returns

// 0 if the strings are the same and returns a number greater than zero

// if the string represented by hLeft_string is bigger than hRight_string.

int my_string_compare(MY_STRING hLeft_string, MY_STRING hRight_string)

{

        int i, asciiSizeLeft = 0, asciiSizeRight = 0;

        char diff_char;

        //cast to known type in order to access object data

        My_string *pLeft_string = (My_string *)hLeft_string;

        My_string *pRight_string = (My_string *)hRight_string;
```

```
	if (pRight_string->size == pLeft_string->size)
	{
		//compares each value
		for (i = 0; i < pRight_string->size; i++)
		{
			diff_char = pLeft_string->data[i] - pRight_string->data[i];
			if (diff_char != 0)
			{
				return diff_char;
			}
		}
		return 0;
	}
	else
	{
		if (pRight_string->size > pLeft_string->size)
		{
			return -1;
		}
		else
		{
			return 1;
		}
	}


}

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: hMy_string will be the handle of a string object that contains
// the next string from the file stream fp according to the following rules.
// 1) Leading whitespace will be ignored.

// 2) All characters (after the first non-whitespace character is obtained

// and included) will be added to the string until a stopping condition

// is met. The capacity of the string will continue to grow as needed

// until all characters are stored.

// 3) A stopping condition is met if we read a whitespace character after

// we have read at least one non-whitespace character or if we reach

// the end of the file.

// Function will return SUCCESS if a non-empty string is read successfully.

// and failure otherwise. Remember that the incoming string may aleady

// contain some data and this function should replace the data but not

// necessarily resize the array unless needed.

Status my_string_extraction(MY_STRING hMy_string, FILE *fp)

{

        int noc;

        char c;


        //cast

        My_string *pstring = (My_string *)hMy_string;

        //set string size to zeroe size to 0 and writes over old content

        pstring->size = 0;


        /*gets first char without white space then checks then

        it has guard in case eof is reached and c is corrupted

        in which case function fails*/

        noc = fscanf(fp, " %c", &c);

        if (noc == EOF)

        {

                return FAILURE;

        }

        if (my_string_push_back(hMy_string, c) == FAILURE)

        {
```

```
		printf("ERROR: Failed to allocate space");
		return FAILURE;
	}

	//reads all spaces after first non white space ends if eOf is reached or a white space
	while (noc == 1 && c != ' ' && noc != EOF && c != '\n')
	{
		/*assumes noc has a healthy value and c is not a white space
		gets a new value checks if the program is at Eof, or picked up another white space or
newline and should return
		if it makes it past that exit condition then it puts the element in the
		array. */
		noc = fscanf(fp, "%c", &c);
		if (noc == EOF)
		{
			return SUCCESS;
		}
		if (c == '\n' || c == ' ')
		{
			break;
		}
		if (my_string_push_back(hMy_string, c) == FAILURE)
		{
			printf("ERROR: Failed to allocate space");
			return FAILURE;
		}
	}

	//moves file pointer back a position so white space is readable
	fseek(fp, -1, SEEK_CUR);
	//makes it here it has should have run succesfuly
	return SUCCESS;
}
```

```
//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Writes the characters contained in the string object indicated
// by the handle hMy_string to the file stream fp.
// Function will return SUCCESS if it successfully writes the string and
// FAILURE otherwise.
Status my_string_insertion(MY_STRING hMy_string, FILE *fp)
{
	My_string *pstring = (My_string *)hMy_string;

	int noc = fprintf(fp, "%.*s", pstring->size, pstring->data);
	return noc >= 0 ? SUCCESS : FAILURE;
}

//Pre-condition: gets a handle to a string and value to add to the string
//Post-condition: item will be added to the string and string->data will be double if
//the data is not within the capacity
Status my_string_push_back(MY_STRING hMy_string, char value)
{
	//casts to known type
	My_string *pstring = (My_string *)hMy_string;
	//creates a temp for data which is a dynamically allocated array for chars
	//char* temp;

	//if there isn't enough space we will make space
	if (my_string_resize_up(hMy_string) == FAILURE)
	{
		return FAILURE;
	}

	//puts char value at the next index then increaeses size
	/*Note: This works because size when no items are in is
	* '0' and when one item is in size is '1' but mind the one
	* item is in index '0' not one so size is already one place
	* ahead as far as indexes are concerend.
	*/
	pstring->data[pstring->size] = value;
	pstring->size++;

	// the program made it to here it had to have run correctly.
	return SUCCESS;
}

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Removes the last character of a string in constant time.
// Guaranteed to not cause a resize operation of the internal data. Returns
// SUCCESS on success and FAILURE if the string is empty.
Status my_string_pop_back(MY_STRING hMy_string)
{
	My_string *pstring = (My_string *)hMy_string;
	if (pstring->size > 0 && pstring != NULL)
	{
		pstring->size--;
		return SUCCESS;
	}

	return FAILURE;
}

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Returns the address of the character located at the given
// index if the index is in bounds but otherwise returns NULL. This address
// is not usable as a c-string since the data is not NULL terminated and is
// intended to just provide access to the element at that index.
```

```
char *my_string_at(MY_STRING hMy_string, int index)
{
	My_string *pstring = (My_string *)hMy_string;
	//guard against inproper bounds
	if (index < 0 || index >= pstring->size)
	{
		return NULL;
	}
	//does address aritmitic fo find nth index
	return pstring->data + index;
}

//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Returns the address of the first element of the string object
// for use as a c-string. The resulting c-string is guaranteed to be NULL
// terminated and the memory is still maintained by the string object though
// the NULL is not actually counted as part of the string (in size).
char *my_string_c_str(MY_STRING hMy_string)
{
	My_string *pstring = (My_string *)hMy_string;
	if (pstring == NULL)
	{
		return NULL;
	}

	//resizes if neccasary
	if (pstring->size >= pstring->capacity)
	{
		if (my_string_resize_up(hMy_string) == FAILURE)
		{
			return NULL;
		}
	}

	pstring->data[pstring->size] = '\0';
	return pstring->data;
}

//precondition: hMystring is a valid object handle
//postcondtion: the memory footprint of hMystring data will be doubled
Status my_string_resize_up(MY_STRING hMy_string)
{
	My_string *pstring = (My_string *)hMy_string;
	char *temp;
	int i;

	if (pstring->size >= pstring->capacity)
	{
		//says take size of char multiply it with the current size of capacity to get
		// current capacity then multiply that by 2 to double size
		temp = (char *)malloc(sizeof(char) * pstring->capacity * 2);
		if (temp == NULL)
		{
			return FAILURE;
		}
		//compy data into temporary
		for (i = 0; i < pstring->size; i++)
		{
			temp[i] = pstring->data[i];
		}
		//frees old array at data but gives us space to add new one
		free(pstring->data);
		pstring->data = temp;
		//doubles capacity to match the new size
```

```
                pstring->capacity *= 2;


                return SUCCESS;
        }


        return SUCCESS;
}


//Precondition: hMy_string is the handle to a valid My_string object.
//Postcondition: Returns an enumerated type with value TRUE if the string
// is empty and FALSE otherwise.
Boolean my_string_is_empty(MY_STRING hMy_string)
{
        My_string *pstring = (My_string *)hMy_string;
        if (pstring->size == 0)
        {
                return TRUE;
        }
        return FALSE;


}


//Precondition: hResult and hAppend are handles to valid My_string objects.
//Postcondition: hResult is the handle of a string that contains the original
// hResult object followed by the hAppend object concatenated together. This
// function should guarantee no change to the hAppend object and return
// SUCCESS if they operation is successful and FAILURE if the hResult object
// is unable to accomodate the characters in the hAppend string perhaps
// because of a failed resize operation. On FAILURE, no change to either
// string should be made.
Status my_string_concat(MY_STRING hResult, MY_STRING hAppend)
{
	My_string *pstring1 = (My_string *)hResult;
	My_string *pstring2 = (My_string *)hAppend;
	Status critical_error;
	int i, size;
	char c;

	//gets size of hresult
	size = pstring1->size;

	//checks for a valid input
	if (pstring1 == NULL || pstring2 == NULL)
	{
		return FAILURE;
	}

	for (i = 0; i < pstring2->size; i++)
	{
		c = pstring2->data[i];

		//segement inserts new letters into the result string
		if (my_string_push_back((MY_STRING)pstring1, c) != SUCCESS)
		{
			while (i > size)//if it failed to push then run if statment till i = original size
			{
				critical_error = my_string_pop_back(hResult);//removes elments form size
till it reaches orginal content
				if (critical_error == FAILURE)//if it failed to pop back all the way then...
				{
					printf("CRITICAL ERROR: STRING CORRUPTED\n");//there was a
critical error and string is corrupted
					return FAILURE;
				}
				i--;
				}
				return FAILURE;
			}
		}
		return SUCCESS;
}



//Precondition:current_word_family, new_key and word are all handles to valid
// MY_STRING opaque objects. guess is an alphabetical character that can be either
// upper or lower case.
//Postcondition: Returns SUCCESS after replacing the string in new_key with the key
// value formed by considering the current word family, the word and the guess.
// Returns failure in the case of a resizing problem with the new_key string.
Status get_word_key_value(MY_STRING current_word_family, MY_STRING new_key, MY_STRING word,
char guess)
{
		int i;
		My_string *pString_current_word_family = (My_string *)current_word_family;
		My_string *pString_word = (My_string *)word;

		my_string_assignment(&new_key, current_word_family);
		My_string *pString_new_key = (My_string *)new_key;

		for (i = 0; i < my_string_get_size(word); i++)
		{
			if (*(my_string_at(word, i)) == guess)
			{
				pString_new_key->data[i] = guess;
			}
		}

		return SUCCESS;
```

```
}
```

UNIT_TEST.H

```
#ifndef UNIT_TEST_H

#define UNIT_TEST_H

#include "my_string.h"



        Status test_init_default_returns_nonNULL(char* buffer, int length);

        Status test_get_size_on_init_default_returns_0(char* buffer, int length);

        Status dbergero_test_get_capacity_on_init_default_returns_7(char* buffer, int length);

        Status dbergero_test_data_allocation_on_init_default_returns_handle(char* buffer, int length);

        Status dbergero_test_my_string_destroy_sets_handle_to_NULL(char* buffer, int length);

        Status dbergero_test_my_string_init_c_string_returns_valid_handle(char* buffer, int length);

        Status dbergero_test_my_string_init_c_string_returns_size_of_string_TEST(char* buffer, int
length);

        Status dbergero_test_my_string_init_c_string_returns_capacity_of_string_TEST(char* buffer, int
length);

        Status dbergero_test_my_string_compare_returns_0_for_same_size(char* buffer, int length);

        Status
dbergero_test_my_string_compare_returns_greater_than_0_for_hleft_greater_than_hright(char* buffer,
int length);

        Status
dbergero_test_my_string_compare_returns_less_than_0_for_hright_greater_than_hleft(char* buffer, int
length);

        Status dbergero_test_my_string_extraction_skips_leading_blanks(char* buffer, int length);

        Status dbergero_test_my_string_extraction_expands_my_string_so_size_less_than_capacity(char*
buffer, int length);

        Status dbergero_test_my_string_extraction_fails_for_empty_string(char* buffer, int length);

        Status dbergero_test_my_string_insertion_sets_size_to_0(char* buffer, int length);

        Status dbergero_test_my_string_insertion_returns_SUCCESS(char* buffer, int length);

        Status dbergero_test_my_string_push_back_resize_SUCCESS(char* buffer, int length);

        Status dbergero_test_my_string_push_back_stores_char_value(char* buffer, int length);

        Status dbergero_test_my_string_at_returns_NULL_at_invalid_index(char* buffer, int length);

        Status dbergero_my_string_at_returns_valid_address(char* buffer, int length);

        Status dbergero_my_string_c_str_returns_valid_address(char* buffer, int length);

    Status dbergero_my_string_c_str_is_NULL_terminated(char* buffer, int length);
```

```
	Status dbergero_my_string_c_is_empty_returns_true_for_empty(char* buffer, int length);

	Status dbergero_my_string_c_is_empty_returns_false_for_nonempty(char* buffer, int length);

	Status dbergero_my_string_concat_successfully_appends_word(char* buffer, int length);


#endif
```

UNIT_TEST.C

```
#include <stdio.h>

#include <string.h>

#include "unit_test.h"


int main(int argc, char* argv[])

{

	Status (*tests[])(char*, int) =

	{

		test_init_default_returns_nonNULL,

		test_get_size_on_init_default_returns_0,

		dbergero_test_get_capacity_on_init_default_returns_7,

		dbergero_test_data_allocation_on_init_default_returns_handle,

		dbergero_test_my_string_destroy_sets_handle_to_NULL,

		dbergero_test_my_string_init_c_string_returns_valid_handle,

		dbergero_test_my_string_init_c_string_returns_size_of_string_TEST,

		dbergero_test_my_string_init_c_string_returns_capacity_of_string_TEST,

		dbergero_test_my_string_compare_returns_0_for_same_size,


	dbergero_test_my_string_compare_returns_greater_than_0_for_hleft_greater_than_hright,

		dbergero_test_my_string_compare_returns_less_than_0_for_hright_greater_than_hleft,

		dbergero_test_my_string_extraction_skips_leading_blanks,

		dbergero_test_my_string_extraction_expands_my_string_so_size_less_than_capacity,

		dbergero_test_my_string_extraction_fails_for_empty_string,

		dbergero_test_my_string_insertion_sets_size_to_0,

		dbergero_test_my_string_insertion_returns_SUCCESS,

		dbergero_test_my_string_push_back_resize_SUCCESS,

		dbergero_test_my_string_push_back_stores_char_value,
            dbergero_test_my_string_at_returns_NULL_at_invalid_index,

            dbergero_my_string_at_returns_valid_address,

            dbergero_my_string_c_str_returns_valid_address,

            dbergero_my_string_c_str_is_NULL_terminated,

            dbergero_my_string_c_is_empty_returns_true_for_empty,

            dbergero_my_string_c_is_empty_returns_false_for_nonempty,

            dbergero_my_string_concat_successfully_appends_word

};


int number_of_functions = sizeof(tests) / sizeof(tests[0]);

int i;

char buffer[500];

int success_count = 0;

int failure_count = 0;


for(i=0; i<number_of_functions; i++)

 {

             if(tests[i](buffer, 500) == FAILURE)

            {

             printf("FAILED: Test %d failed miserably\n", i);

            printf("\t%s\n", buffer);

            failure_count++;

            }

            else

            {

            // printf("PASS: Test %d passed\n", i);

            // printf("\t%s\n", buffer);

            success_count++;

            }

}


printf("Total number of tests: %d\n", number_of_functions);
```

```
	printf("%d/%d Pass, %d/%d Failure\n", success_count,
	number_of_functions, failure_count, number_of_functions);

	return 0;
}
```